CMPT 280

Topic 5: Abstract Data Types (ADTs)

Mark G. Eramian

University of Saskatchewan

References

- Textbook, Chapter 5

Reading Referesher: Quiz

1. What is a data structure?
2. What is the difference between a data type and a data structure?
3. What is an abstract data type?
4. Why do we use ADTs?
5. What is the difference between specification and implementation?

A List ADT

Name: List<G>

Sets:
L : set of lists containing items from G
G : set of items that can be in the list
B : $\{\mathbf{true}, \mathbf{false}\}$

Signatures:
newList<G> : $\to L$
L.isEmpty: $\to B$
L.insertFirst(g): $G \to L$
L.firstItem: $\not\to G$
L.deleteFirst: $\not\to L$

Preconditions: For all $l \in L$, $g \in G$
newList<G>: none
l.isEmpty: none
l.insertFirst(g): none
l.firstItem: l is not empty
l.deleteFirst: l is not empty

Semantics: For $l \in L$, $g \in G$
newList<G>: construct new empty list to hold elements from G.
l.isEmpty: return true if l is empty, false otherwise
l.insertFirst(g): g is added to l at the beginning.
l.firstItem: returns the first item in l.
l.deleteFirst: removes the first item in l.

Note: the symbol $\not\to$ denotes a partial function.

A Student ADT

Name: Student

Sets:
S : set of all students
N : set of all names
K : set of all student numbers

Signatures:
newStudent(n, k) : $N \times K \to S$
S.getName: $\to N$
S.getNumber: $\to K$
S.setName(n): $N \to S$
S.setNumber(k): $K \to S$

Preconditions: For all $s \in S$, $n \in N$, $k \in K$
newStudent: none
s.getName: none
s.getNumber: none
s.setName(n): none
s.setNumber(k): none

Semantics: For $s \in S$, $n \in N$, $k \in K$
newStudent(n, k): construct new student with name $n \in N$ and number $k \in K$
s.getName: return name of s
s.getNumber: return student number of s
s.setName(n): change name of s to n.
s.setNumber(k): change student number of s to k.

Exercise 1: ADT for a Queue with Bounded Size

Fill in the blanks

Name: Queue$<G>$

Sets:

Q : set of queues containing items from G
G : set of items that can be in the queue
B : $\{\mathbf{true}, \mathbf{false}\}$
$\mathbb{N}_0$: set of non-negative integers

Signatures:

newQueue$<G>(n)$:
Q.isEmpty:
Q.isFull:
Q.add(g):
Q.remove:

Preconditions: For all $q \in Q$, $g \in G$, $n \in \mathbb{N}_0$

newQueue$<G>(n)$:
q.isEmpty:
q.isFull:
q.add(g):
q.remove:

Semantics: For $q \in Q$, $g \in G$, $n \in \mathbb{N}_0$

newQueue$<G>(n)$: create a queue of items from G with capacity n
q.isEmpty: returns true if q is empty, false otherwise
q.isFull: return true if q is full, false otherwise
q.add(g): enqueues g at the back of the queue
q.remove: removes then returns the item at the front of the queue

Exercise 2

S.pop() -/-> S

S.top() -/-> G

S.push(g) G -/-> S

S.isEmpty() & S.isFull() -> B

- Write an ADT for a stack with bounded size.

Specification vs. Implementation

- Specifications do not say anything about the implementation, only the interface.
- We did not specify the specific pieces of data the ADT will need.
    - Student ADT: no specific types for name and student number.
    - List ADT: no head or tail fields for the list.
    - Queue ADT: no fields to keep track of the capacity of the queue or the number of elements it contains.
- We did not specify any underlying data structures:
    - List could be arrayed or linked.
    - Queue could use array, list.
- All of these decisions are saved for the implementation.

Implementing ADTs in Java

- Choose data types for sets.
- Type parameters remain type parameters for the class.
- Signatures become class methods.
- Preconditions become `@precond` entries for Javadoc (which the method's code should verify before proceeding) and also manifest as specific regression tests to make sure the code that verifies the preconditions operates correctly.
- Sematics become algorithms.

List Specification $\rightarrow$ Implementation

Defining the class and methods

Signatures:

newList$\lt G \gt$: $\rightarrow L$
L.isEmpty: $\rightarrow B$
L.insertFirst(g): $G \rightarrow L$
L.firstItem: $\not\rightarrow G$
L.deleteFirst: $\not\rightarrow L$

```java
public class List<G> {
    public List() {}
    public boolean isEmpty() {}
    public void insertList(G e) {}
    public G firstItem() {}
    public void deleteFirst() {}
}
```

Observe that if the signature contains $\rightarrow L$ and just modifies the state of a list rather than making a new one, then the return type is `void`.

Intermezzo: ADT Specification is Language Independent!

Signatures:

newList$<G>$: $\to L$

L.isEmpty: $\to B$

L.insertFirst(g): $G \to L$

L.firstItem: $\not\to G$

L.deleteFirst: $\not\to L$

```
class List:
    def __init__(self):
        ...
    def isEmpty(self):
        ...
    def insertList(self, e):
        ...
    def firstItem(self):
        ...
    def deleteFirst(self):
        ...
```

Python

```
template<typename G>
class List {
public:
  List() {}
  bool isEmpty() {}
  void insertList(G e) {}
  G firstItem() {}
  void deleteFirst() {}
};
```

C++

```
typedef ... G;
typedef struct { ... } List;

List *createList();
int isEmpty(List *l);
void insertList(List *l, G e);
G firstItem(List *l);
void deleteFirst(List *l);
```

C

List Specification → Implementation

Adding the Precondtions

Preconditions: For all $l \in L$,
$g \in G$
newList$<G>$: none
l.isEmpty: none
l.insertFirst(g): none
l.firstItem: l is not empty
l.deleteFirst: l is not empty

```
public class List<G> {
    public newList() {}
    public void isEmpty() {}
    public void insertList(G e) {}
    public G firstItem() {}
    public void deleteFirst() {}
}
```

```
public class List<G> {

    public newList() {}

    public void isEmpty() {}

    public void insertList(G e) {}

    /**
     * @precond The list is not empty.
     */
    public G firstItem() {}

    /**
     * @precond The list is not empty.
     */
    public void deleteFirst() {}
}
```

List Specification → Implementation

Checking the Preconditions.

Methods that have a precondition are written to throw an exception if the precondition isn't true, for example:

```
    /**
     * Get the item at the front of the list
     * @precond The list is not empty.
     * @throws InvalidStateException if the list is empty.
     */

     public G firstItem() throws ContainerEmpty280Exception {
        // verify the precondition
        if( this.isEmpty() )
            throw new ContainerEmpty280Exception();

        // Rest of implementation...
    }
```

List Specification → Implementation

Adding the Semantics

```java
public class List<G> {
    /**
     * Create a new list
     */
    public newList() {}

    /**
     * Test whether the list is empty.
     * @return true if the list is empty, false otherwise.
     */
    public void isEmpty() {}

    /**
     * Add an item to the beginning of the list.
     * @param e     the item to be added to the list.
     */
    public void insertList(G e) {}

    /**
     * Get the item at the front of the list
     * @precond The list is not empty.
     * @return the item at the beginning of the list.
     */
    public G firstItem() {}

    /**
     * Delete the first element of the list.
     * @precond The list is not empty.
     */
    public void deleteFirst() {}
}
```

List Specification → Implementation

Add the data structures and method implementations.

```
public class List<G> {

    // Let's use a linked list
    // Of course, ListNode is an implementation of
    // a ListNode ADT which we would have to specify...
    protected ListNode<G> head;

    /**
     * Create a new list
     */
    public List() { head = null; }

    ...
}
```

```
public class List<G> {

    // Or maybe it's an array...
    protected G[] listItems;


    /**
     * Create a new list
     */
    public List() { listItems = (G[]) new Object[100]; }

    ...
```

Summary

- ADT specification describes a type independent of the implementation language.
- Uses set and function notation to achieve this independence.
- Set definitions, operations, preconditions and semantics are given.
- ADT can be implemented in any language.

Next Class

- Next class reading: Chapter 6: Trees.